Developing Secure Objects

Deborah Frincke
Department of Computer Science
University of Idaho

Keywords: Object-oriented design, development, security policy, COTS

Abstract

Distributed object systems are increasingly popular, and considerable effort is being expended to develop standards for interaction between objects. Some high-level requirements for secure distributed object interaction have been identified. However, there are no guidelines for developing the secure objects themselves. Some aspects of object-oriented design do not translate directly to traditional methods of developing secure systems. In this paper, we identify features of object oriented design that affect secure system development. In addition, we explore ways to derive secure object libraries from existing commercial off-the-shelf (COTS) class libraries that lack security, and provide techniques for developing secure COTS libraries with easily modifiable security policies.

Introduction

Object-oriented design (OOD) techniques have become one of the more popular methods used in designing software systems. Some reasons for this popularity are perceived benefits such as a high potential for software reuse, improved reliability, and lower developement and maintenance costs. In particular, distributed system design seems especially amenable to an object-oriented approach. Object-oriented programming (OOP) languages also appear to be very appropriate candidates for use in secure system development. OOP languages usually support information hiding, which assists in designing software components that separate policy and mechanism and more reliable software.

There are, however, pitfalls. Object-oriented systems are designed using individual objects that are actively responsible for maintaining their own integrity. It might not be appropriate to design such systems using the traditional 'security monitor' paradigm. Newer OOD techniques, such as Gamma's design patterns, use dynamic modification of system components, where functionality can change substantially as the system runs. If components can change functionality dynamically, it may become quite difficult to validate or ensure maintenance of security policies. Object-oriented systems are intended to be assembled using pre-existing components, and these components may not have been developed for use in secure environments. Potentially, every object in a distributed object system is vulnerable to attack or misuse. Thus, secure system designers must pay particular attention to the security-relevant attributes of the objects they use. Secure system designers who wish to take advantage of OOD will need to address these issues.

In this paper, we identify some features of OOD that affect secure system development. We also explore ways to derive secure object libraries from existing commercial off-the-shelf (COTS) class libraries that lack security, and techniques for developing secure COTS libraries.

Background

Many features of OOP languages, such as their support for information hiding, data encapsulation, and separate components, make them seem well suited for use in secure system development. OOD techniques potentially can make it easier to separate software system's security policy from the mechanisms used to enforce it. Systems with a high degree of modularity should be easier to validate than those without it. Undesirable information flow should be easier to prevent by data hiding techniques. In this section, we describe standards and issues for secure systems, discuss what OO developers have done to address these issues to date, then describe some features of OOPs in general and C++ in particular that are relevant to system security.

TCSEC

The Department of Defense' Trusted Computer Security Criteria (TCSEC), was developed to serve as a guideline for secure system developers, as a way to specify assurance requirements for procurement, and to provide an assessment standard for secure systems. TCSEC descriptions refer to an abstract *security monitor*, so access to system resources is (abstractly at least) guarded by a centralized monitor. This abstract description has been used in implementing secure systems, partially because it is easier to validate correct resource access control with this design. However, if individual objects are actively responsible for maintaining their own integrity, the mechanisms responsible for enforcing object access policies are likely to be distributed throughout the system, rather than gathered in a centralized location. This distribution may make the validated assurance requirements more difficult to meet.

CORBA

One emerging standard for distributed systems is the Common Object Request Broker Architecture, or CORBA. The CORBA standard has been developed through the Object Management Group (OMG) Consortium, which includes over 500 members. CORBA's Object Management Architecture (OMA) describes how end-user application objects, object services, and general purpose services interact together within a distributed object system. Cooperation is achieved through the Object Request Broker (ORB), which enables objects to transparently send and receive requests to other objects [MZ95].

At the time of this writing, the CORBA standard itself discusses security only briefly, although this is expected to change in early 1996. CORBA requires authentication of object clients but does not describe how this will take place [MZ95]. The Basic Object Adapter is responsible for defining how objects are activated (shared server, unshared server, server-per-method, and persistent server), and includes the following five functions as described in [OHE95]: *Authentication* of object clients, although the style (and trustworthiness) of this authentication is not defined, *Implementation Repository* for installation and registry of objects and object descriptions, *Mechanisms* for object activation/deactivation, communication with objects (including parameter passing), and generating/interpreting object references, and *Activation/Deactivation* and *Method Invocation* of implementation objects.

In 1994 the OMG issued a *White Paper on Security* to address the issue of security in distributed object systems and provide guidance to OMG members in their development of proposals for security in CORBA-compliant object systems [Gro94]. This white paper outlines aspects of distributed object systems which can affect the security of the overall system, such as the number of components, complexity of component interactions, and lack of clear-cut boundaries of trust. Specific security areas of interest include confidentiality, integrity, accountability, and availability; functional security requirements include identification and authentication, authorization and access control, security auditing, secure communication, cryptography, and administration. OMG is reviewing several proposals from vendors such as Hewlett-Packard and Sun.

Our work evaluates methods of implementing security issues such as those discussed in the CORBA security standards. We identify general principles for secure object development which are expected to be important both for customized security components that do not adhere to CORBA standards and for components that are developed for use within CORBA.

Low level design effects

Design perspective is of particular concern because it has such significant effects on policy, and these decisions are made by class developers (and thus unlikely to be the identical between competing class libraries). Changing libraries or adding objects from new libraries may result in a different access control policy. Secure system designers cannot take advantage of "interchangeable components" without carefully considering the details of the design strategies used in developing the class libraries they use. In this section, we summarize ways in which decisions made by low-level component designers can change the security policy of the system.

Identifying subjects and objects Most access control models require the subject to have a security label that is compared to that of the object. The Bell and LaPadula access control model (BLP), a subject may obtain data (read from) an object only if subject's access control label dominates[1] that of the target object. If the subject wishes to write to a target object, the *object's* label must dominate that of the subject. In traditional operating system activities, it is clear which participant is the subject and which is the object. This determination can usually be made based on the *type* associated with the participant. When a process requests information from a file, the file is the object and the process is the subject. Files will rarely if ever be subjects, though processes may be either subjects or objects. Due to the emphasis on active objects OOD techniques require, traditionally passive objects such as files may become subjects in OOD systems. Designers must determine whether a `File` object that *updates itself* and *provides data about itself* is really the same as a passive file that is written to or read from.

Placement of enforcement mechanisms Consider a file containing records, each with a security label. Traditionally, access checks take place at the file level, perhaps handled by a file monitor. A more active design places responsibility with the record, as the object most closely concerned. Rather than encapsulating policy enforcement in a single external monitor, in OOD we are more likely to distribute both the responsibility for enforcing policy and the mechanisms for enforcing policy. Thus, low-level object designers will take on responsibility for system security design. For example, consider a file containing records, each with a security label. When clients request records from the file, should the *file* authenticate and validate the request before producing the requested

[1] *Dominates* here means that in the universe of access control labels under discussion there is a partial ordering $O \prec S$ between some object and subject labels, and the rest are unrelated. Usually equality of labels is included.

Activity	**Result**
Lists copy themselves at client's behest. Unlabelled elements.	Only elements readable by client
Lists copy themselves at their own request.	Identical lists
Unlabelled lists pass copy request from client to labelled elements, which copy themselves	Only elements readable by client
Unlabelled lists tell list elements to insert themselves in the new list	Identical lists
Labelled lists pass on copy request from client to to labelled elements, which insert themselves in the new list	Only elements at the same level as the client

Figure 1: Copied object characteristics

items, or should the *records* decide whether a client may perceive their value? If data is to be written to the file on the behalf of a user, do we consider the data's label or the user's label?

Copy operations OOD places `copy` operations within class definitions. An OOD list `copy` is designed so that the list being copied is responsible for performing the copy itself. Table shows how result of a `copy` operation changes depending on whether the subject is considered to be the list being copied or the client requesting the copy operation. This in turn depends on how 'active' the list's elements are. This decision would ideally be consistent for all components used within a particular system, but that is unlikely to be the case if these components are obtained from different COTS libraries.

Aggregations When low-level objects are responsible for determining the extent of client's access to their contents, designers will need to take care to shield the existence of some objects from clients, lest they introduce a covert channel. A simple example involves a list, a client, and a set of list elements. The list's designer must balance OO design strategies (making objects responsible for managing their own contents) with security needs (such as preventing inappropriate information flow). Suppose that the designer decides to create a secure list. One option is to give the list's elements control over their own values. Suppose that a client object requests information from a list element. Since the element object is active, the element may refuse to supply information, but cannot prevent the client object from learning of its existence (potential covert channel). An alternate design may be used to eliminate this covert channel. Secure lists may be written so that not only do the list elements know their own security labels, but the list is the object that determines whether the client will know if an individual list element exists at all.

An advantage of placing access controls at a low level (say, with data elements) is that elements may be passed between clients without concern. If policy enforcement is at too high a level (say, file level), then the client must be trusted not to pass high security records to low security subjects. If the security policy enforcement is lower, then the client object need not be trusted.[2]

[2] Assuming objects really do control their own integrity and information flow. In languages that allow programmers to bypass the high-level visibility constraints, for example using pointers, additional methods are needed. Objects might include digital signatures to prove that they have not been modified in transit, and information might be encrypted.

Existing libraries

If OOD is to be fully exploited by secure system developers, they should not expect to design all of their system objects in-house. However, if externally developed libraries are used, it will be necessary to add security features to the classes they contain. In this section we explore ways to derive secure object libraries from existing commercial off-the-shelf (COTS) class libraries that lack security.

We consider a collection of interacting objects to be a system. Objects within the system may be active or passive or both. Thus, we need to define both an object system security policy and provide a set of object system security mechanisms. General goals for secure object design follow:

1. Attach an identity to each object and object client
2. Separation of policy and mechanism
3. Separation of object abstractions and security requirements
4. Information Flow and Covert Channels

Without supplying a unique and immutable identity for each object in a distributed system, it may prove difficult to reliably determine whether the interactions an object requests should be allowed. Proper identification and authentication is required for most levels of secure systems.

Separation of policy and mechanism is important for several reasons. There is no single "correct" access control policy that can be predetermined. If components are to be reused in different systems, it is important to make policy modifications easy to perform. Second, there is no single "correct" set of mechanisms that can be determined in advance. Some mechanisms are expensive, unavailable, or illegal.[3] Finally, changing mechanisms can effectively change policies, and this would be unacceptable in most secure systems.

If object abstractions are too closely connected to system security requirements, maintaining the system will become quite difficult. First, class designers will be required to become cognizant of security concerns. Second, changing security requirements might result in forced changes to all classes in the library, which would add considerable expense as well as increasing the likelihood of error. Finally, validation would become harder, since security requirements would be hard to distinguish from object characteristics.

Direct modification

We can create secure objects adding security features to their definition. Although direct modification should not really be considered a component reuse technique, we include it here for completeness and comparison. Direct modification should ensure at least the following: the object has an appropriate sensitivity label, all public members manipulate data in accordance with that label and the label of the client object. In our example, the functions that manipulate the `File` objects data should be declared `private`, and hence invisible outside `File`. `Public` functions, `readSecureFile()` and `writeSecureFile`, would be made available for use by `File` clients. This

[3]Ex: certain forms of cryptography used in authentication mechanisms are illegal to use or export in some countries, including the USA.

method is simple and efficient, and the security policy associated with each object is easy to determine and modify.

Unfortunately, since the security policy is embedded within the class, changing the system security policy implies changing all class descriptions, as does a change in the choice of security mechanisms. Validating this security policy implementation will be time consuming, since we must examine every class. The security mechanism is necessarily visible to clients,[4] and we are assuming that the calling object will properly supply the client's identity.[5] In this particular example, `File` data will not retain the original security label once passed to a client, making it difficult to enforce fine-grained information flow controls or to retain access control information across boundaries.

It is often impractical or impossible either to modify class descriptions or to create secure versions. The former may cause existing software to break, and the latter will certainly increase application code size (as well as being tedious!).

Wrappers

We can use wrappers to provide a secure interface to objects that are not already This technique is expected to be most useful when portions of the class definition or implementation cannot be changed or are invisible, as with proprietary or compiled libraries. In this technique, we use an existing library of objects that does not include security information, and supply a wrapper class that contains security-relevant data such as identifiers and sensitivity labels as well as enforcement mechanisms.

`SecureFile` is defined using private inheritance from the `File` base class, which makes `File` members invisible outside `SecureFile`, and adds operations that manipulate the base class securely. `SecureFile` clients cannot manipulate the file unless they pass the authentication test supplied by the wrapper. The COTS vendor can modify the `File` base class without affecting the security policies or mechanisms of the specialized class. In particular, `File`'s designer need not consider whether `File` objects will be used in a secure environment. Additional member functions could safely be added to `File`, e.g., `char *appendFile(char *data)`. Because private inheritance is used in creating the secure class, such additional member functions will not be available to users of `SecureFile` unless the `SecureFile` designer adds members that use them, and so the read/write policy of `SecureFile` cannot be bypassed.

There are still disadvantages. We cannot authenticate `SecureFile` clients without adding wrappers to those clients as well, and `SecureFile`'s designer must be cognizant of changes in the underlying class `File` to maintain equivalent functionality. This latter point would not be an issue if we had used public or protected inheritance from the base class rather than private, but the first of these would permit clients of `SecureFile` to bypass the security mechanisms, and the second would let descendants of `SecureFile` bypass those mechanisms. This method is somewhat more suitable for communication between trusted objects and untrusted objects, since untrusted objects cannot manipulate trusted data directly without backwards type casting of the secure object.

Multiple base classes

[4]The implementation code is visible unless we pre-compile member function code and only make the class definition header visible. While "security by obscurity" is insufficient for data protection, sometimes it is desirable to hide security mechanisms. That is not possible here unless the entire class definition is hidden.

[5]Acceptable within a trusted object base, unacceptable for untrusted objects.

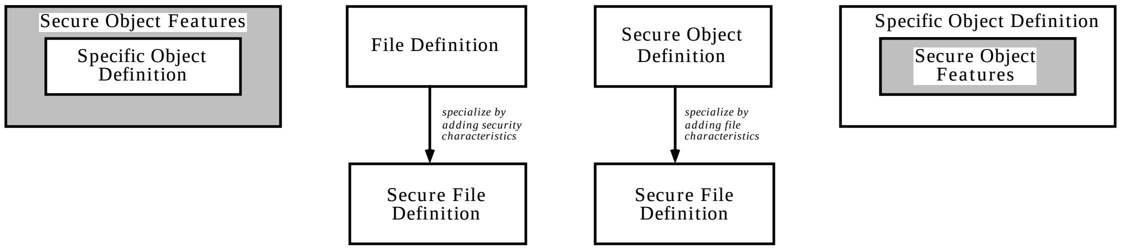
Secure Object Features
Specific Object Definition
File Definition
specialize by adding security characteristics
Secure File Definition
Secure Object Definition
specialize by adding file characteristics
Secure File Definition
Specific Object Definition
Secure Object Features

Figure 2: Reversing Inheritance Order

In this technique, we use an existing library of objects that does not include security information (`BaseObject`), and supply a class that contains security-relevant data such as identifiers and sensitivity labels (`CommonSecurityInfo`). We then generate a secure class `SecureBaseObject` by inheriting both from `BaseObject` and `CommonSecurityInfo`. The resultant class must re-define all methods that manipulate or observe `BaseObject` with new methods that make use of the security data in `CommonSecurityInfo`. Inheritance from `BaseObject` must be private. Figure 3 provides an example, using `File` as the unsecured COTS component and `SecureFileDefinition` as the secured version. This technique is a variant of Gamma's `Adapter` [GHJV94].

The advantage of using multiple inheritance is that the designer of the object which we are securing (`File`) does not need to consider security aspects, which are left to the developer of the security information class (which might include access and integrity labelling) and the developer of the secure file task (which provides manipulators). It is very easy to change security policies and mechanisms, since these changes are locallized in `CommonSecurityInfo`.

A remaining disadvantage is that the `BaseObject` component does not have any protection from "backwards type casting" of `SecureBaseObject`. Hence, it will not be suitable for transmission of objects between security domains that do not trust one another. Transmission between trusted systems along a trusted path, should, however, be acceptable.

Parameterizing classes

One of C++'s newer features, templates, is useful for for maintaining policy/mechanism consistency throughout a system. Figure 6 parameterizes security policy and mechanism in the template base class. We use the template to create system objects that will have identical security policy and enforcement, for example `Secure<File>` and `Secure<Directory>`. There are some restrictions. First, base class designers (e.g., `File`) must use the member names specified in the template (`read` and `write`). Second, the security template object is unlikely to contain all operations of the base classes (or else it would be overly specific) and thus users of the secure objects will have access to less functionality than clients of the original versions. This may make migration to a secure platform difficult. Since the template feature of C++ is relatively new, its flexibility may increase. Languages such as OBJ [GM82] permit the programmer to define requirements for parameter characteristics in the template [AFL90]. The object used as a parameter is required to provide certain operations, and so the parameterized class can rely on their presence. OBJ also permits *properties* of object parameters to be specified in this way; if both of these features were added to C++ it

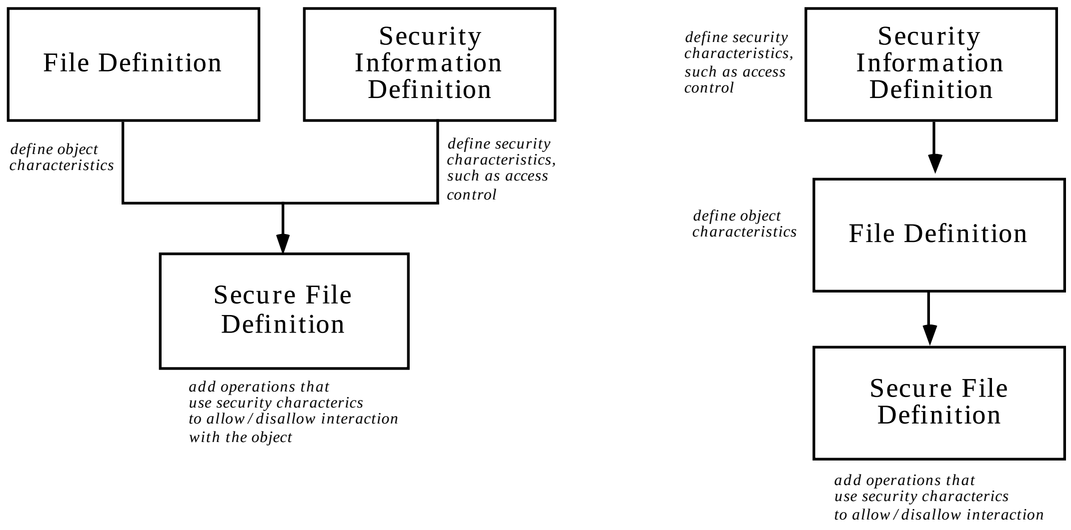
File Definition
Security Information Definition
define object characteristics
define security characteristics, such as access control
Secure File Definition
add operations that use security characterics to allow/disallow interaction with the object
define security characteristics, such as access control
Security Information Definition
define object characteristics
File Definition
Secure File Definition
add operations that use security characterics to allow/disallow interaction

Figure 3: Class combinations

would be a considerable assistance in migrating objects to secure environments.

Common base class

Both the wrapper and template techniques add security as an outer layer, rather than as an integral component of the base abstraction. Security-relevant information and enforcement mechanisms can be removed from objects simply by using type coercion to revert to the base class. The coerced object would lose the security label and it would no longer be marked trusted. However, object data could be viewed without passing any security checks.[6]

An alternate approach includes the security information in a common base class (Figure 2). It will then be impossible to remove the security information. We could thus use our base class with all of the security-relevant information and operations as the parent of all secure objects, creating a uniform security policy. Changing mechanisms would still only require changing the base class method implementations and data.

This technique can also be used with COTS libraries as long as the library developer uses a template as a common base class. Developers can then instantiate objects using a locally defined secure base class. However, care must be taken by the library developers as well as the library users: all object modifications and information flow must be handled using the functionality provided by the secure object base class rather than manipulating the derived class components directly. The wrapped technique provides a barrier between the object's clients and the object's data, and protects against security hazards purposely or inadvertently allowed by the class designer (much as a firewall does for networked systems). The secure base class technique does not, but instead provides functionality that permits secure access. Thus, it cannot prevent misuse of the object, but it can be used to ensure appropriate and consistent use of the object. Combining these techniques increases flexibility and protection, but increases overhead.

[6]This was not a problem when we used direct modification.

Wrappers and base classes

If the COTS library designer has used the Common Base Class technique from Section , then we can further refine our technique providing a second layer of specialization. We are assuming that we have a `SecureBaseObject` that inherits security-relevant data from a `CommonSecurityInfo` object. We then use private inheritance to specialize this `SecureBaseObject` further, by providing a wrapper that handles all access to the object's data. Figure 3 (rightmost) provides an example.

The advantage of using two levels of inheritance is that the designer of the object which we are securing (`File`) can take direct advantage of the security characteristics in writing `File` operations. Thus, these operations can potentially be more efficient than those developed using multiple inheritance. The outer wrapper of inheritance can be used for operations such as authentication between objects, communication protocols, and so on; these are elements that might vary between systems and are not really an inherent part of most objects in the way that a sensitivity label is.

Discussion

Some differences between traditional system development and secure system development are advantageous. When we place access controls with low-level data elements, these elements may be passed among clients without concern. In the best case scenario, they retain the policies desired by their original designer even when transferred between systems. If policy enforcement is at a higher level (say, file level), then the client must be trusted not to pass high security records to low security subjects. Multilevel Secure System (MLS) developers are already required to ensure that entities passed between components retain accurate labelling and are transferred in accordance with the system security policy; this would be a natural result of an OO strategy.

Different security requirements and system environments will affect which of the techniques we have discussed will be most appropriate. We believe that the wrapper technique will prove to be more useful when underlying objects aren't necessarily trusted, and base class technique will be more efficient when the developer can be trusted to use the underlying security mechanisms. Static inheritance of labels may be needed when objects cannot be permitted to change labels; and dynamic labelling should only be permissible when systems can be trusted not to make improper changes. This paper is intended to serve as a starting point for secure system designers who want to begin using object oriented techniques.

References

[AFL90] M. Archer, D. Frincke, and K. Levitt. A template for rapid prototyping of operating systems. *International Workshop on Rapid System Prototyping*, June 4-7 1990.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1994.

[GM82] J. Goguen and J. Meseguer. Rapid prototyping in the OBJ executable specification language. *ACM SIGSOFT Software Engineering Notes*, 7(5):75–84, December 1982.

[Gro94] OMG Security Working Group. OMG white paper on security. Technical report, Object Management Group, April 1994.

[MZ95] T. Mowbray and R. Zahavi. *The Essential CORBA*. John Wiley and Sons, 1995.

[OHE95] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley and Sons, 1995.

C++ Code for Examples Used In Text

```
class File {
public:
     File( char *name, ...) ;
     ~File() ;
    char *readFile() ;
    void writeFile(char *data) ;
private:
     char * data ;
     ...
}
```

```
class SecureFile {
public:
    SecureFile( char *name, ...) ;
    ~SecureFile() ;
    char *readSecureFile(char *reader,...)
         { if reader==owner then return readFile(...) ; }
    void writeSecureFile(char *writer,....,char *data)
         { if writer==owner then writeFile(...) ; }

private:
     char * data ;
     char * owner ;
     char * otherSecurityInfo ;
     char *readFile() ;
     writeFile(char *data) ;
     ...
}
```

Figure 4: Adding security directly.

```
class SecureFile : private File {
public:
    SecureFile( char *name, char *owner, ...) ;
    ~SecureFile() ;
    char *readSecureFile(char *reader,...)
        { if reader==owner then return readFile(...) ; }
    void writeSecureFile(char *writer,....,char *data)
         { if writer==owner then writeFile(...) ; }

private:
      char * owner ;
      char * otherSecurityInfo ;
...
}
```

Figure 5: Using Specialization.

```
template class Secure<base> : private <base> {
public:
    Secure<base>( char *name, char *owner, ...) ;
    ~Secure<base>() ;
    char *readSecure<base>(char *reader,...)
        { if reader==owner then return <base>::read(...) ; }
    void writeSecure<base>(char *writer,....,char *data)
         { if writer==owner then <base>::write(...) ; }
private:
    char * owner ;
    char * otherSecurityInfo ;}
Secure<File>   somefile ;
```

Figure 6: Template instantiation.